Parallel Non-blocking Deterministic Algorithm for Online Topic Modeling

Oleksandr Frei[1] and Murat Apishev[2]

[1] Moscow Institute of Physics and Technology, oleksandr.frei@gmail.com
[2] National Research University Higher School of Economics, great-mel@yandex.ru

Abstract. In this paper we present a new asynchronous algorithm for learning additively regularized topic models and discuss the main architectural details of our implementation. The key property of the new algorithm is that it behaves in a fully deterministic fashion, which is typically hard to achieve in a non-blocking parallel implementation. The algorithm had been recently implemented in the BigARTM library (`http://bigartm.org`). Our new algorithm is compatible with all features previously introduced in BigARTM library, including multimodality, regularizers and scores calculation. While the existing BigARTM implementation compares favorably with the alternative packages such as Vowpal Wabbit or Gensim, the new algorithm brings yet further improvements in CPU utilization, memory usage, and spends even less time to achieve the same perplexity.

Keywords: probabilistic topic modeling, Probabilistic Latent Sematic Analysis, Latent Dirichlet Allocation, Additive Regularization of Topic Models, stochastic matrix factorization, EM-algorithm, online learning, asynchronous and parallel computing, BigARTM.

1 Introduction

Topic models [1] is a powerful machine learning technology for statistical text analysis that has been widely used in text mining, information retrieval, network analysis and other areas [2]. Today a lot of research efforts around topic models is devoted to distributed implementations of *Latent Dirichlet Allocation* (LDA) [4], a specific Bayesian topic model that uses Dirichlet conjugate prior. This lead to numerous implementations such as AD-LDA [7], PLDA [8] and PLDA+ [9], all designed to run on a big cluster. Largest topic models of web scale can reach millions of topics and vocabulary words, yielding Big Data models with trillions of parameters [3]. Yet not all researchers and application are dealing with so large web-scale collections, and they require an efficient implementation that can run on a powerful workstation or even laptop. Such implementations are also very useful, as shown by popular open-source packages Vowpal Wabbit [11], Gensim [10] and Mallet [12], which are neither distributed nor sometimes even multi-threaded.

Scaling down a distributed algorithm can be challenging. LightLDA [3] is a major step in this direction, however it is focused on the LDA model. Our goal is to develop a flexible framework that can learn a wide variety of topic models.

BigARTM [17] is an open-source library for regularized multimodal topic modeling of large collections. BigARTM is based on a novel technique of additive regularized topic models (ARTM) [14,15,16], which gives flexible multi-criteria approach to probabilistic topic modeling. ARTM includes all popular models such as LDA [4], PLSA [5], and many others. Key feature of ARTM is that it provides a cohesive framework that allows users to combine different topic models that previously did not fit together.

BigARTM is proven to be very fast comparing to the alternative packages. According to [17], BigARTM runs approx. 10 times faster comparing to Gensim [10] and twice faster than Vowpal Wabbit (VW) [11] in a single thread. With multiple threads BigARTM wins even more as it scales linearly at least up to 16 threads. In this paper we address the remaining limitations of the library, including few performance bottlenecks and fixing non-deterministic behavior of the Online algorithm.

The rest of the paper is organized as follows. In section 2 we introduce basic notation used throughout this paper. In sections 3, 4, 5 we summarize Offline, Online and Asynchronous Online algorithms for learning ARTM models. In section 6 we compare the internal architecture of BigARTM library between versions v0.6 and v0.7. In section 7 we report results of our experiments on large datasets. In section 8 we discuss advantages, limitations and open problems of BigARTM.

2 Notation

Let D denote a finite set (collection) of texts and W denote a finite set (vocabulary) of all terms from these texts. Let n_{dw} denote the number of occurrences of a term $w \in W$ in a document $d \in D$; n_{dw} values form a sparse matrix of size $|W| \times |D|$, known as *bag-of-words* representation of the collection.

Given an (n_{dw}) matrix, a probabilistic topic model finds two matrices: $\Phi = \{\phi_{wt}\}$ and $\Theta = \{\theta_{td}\}$, of sizes $|W| \times |T|$ and $|T| \times |D|$ respectively, where $|T|$ is a used-defined number of *topics* in the model. Matrices Φ and Θ provide a compressed representation of the (n_{dw}) matrix:

$$n_{dw} \approx n_d \sum_{t \in T} \phi_{wt}\theta_{td}, \text{ for all } d \in D, w \in W,$$

where $n_d = \sum_{w \in W} n_{dw}$ denotes the total number of terms in a document d.

To learn Φ and Θ from (n_{dw}) an additively-regularized topic model (ARTM) maximizes the log-likelihood, regularized via an additional penalty term $R(\Phi, \Theta)$:

$$\sum_{d \in D} \sum_{w \in W} n_{dw} \ln \sum_{t \in T} \phi_{wt}\theta_{td} + R(\Phi, \Theta) \;\to\; \max_{\Phi,\Theta}. \tag{1}$$

Regularization penalty $R(\Phi, \Theta)$ may incorporate external knowledge of the expert about the collection. With no regularization ($R = 0$) it corresponds to

Algorithm 1: Offline algorithm

Input: collection D;
Output: matrix $\Phi = (\phi_{wt})$;

initialize (ϕ_{wt});
create batches $D := D_1 \sqcup D_2 \sqcup \cdots \sqcup D_B$;
repeat
  $(n_{wt}) := \sum\limits_{b=1,\dots,B} \sum\limits_{d \in D_b} \mathsf{ProcessDocument}(d, \Phi)$;
  $(\phi_{wt}) := \operatorname*{norm}\limits_{w \in W}(n_{wt} + \phi_{wt} \frac{\partial R}{\partial \phi_{wt}})$;
until (ϕ_{wt}) *converges*;

PLSA [5]. Many Bayesian topic models, including LDA [4], can be represented as special cases of ARTM with different regularizers R, as shown in [15,16].

In [14] it is shown that the local maximum (Φ, Θ) of the problem (1) satisfies the following system of equations:

$$p_{tdw} = \operatorname*{norm}_{t \in T}\bigl(\phi_{wt}\theta_{td}\bigr); \tag{2}$$

$$\phi_{wt} = \operatorname*{norm}_{w \in W}\left(n_{wt} + \phi_{wt}\frac{\partial R}{\partial \phi_{wt}}\right); \quad n_{wt} = \sum_{d \in D} n_{dw} p_{tdw}; \tag{3}$$

$$\theta_{td} = \operatorname*{norm}_{t \in T}\left(n_{td} + \theta_{td}\frac{\partial R}{\partial \theta_{td}}\right); \quad n_{td} = \sum_{w \in d} n_{dw} p_{tdw}; \tag{4}$$

where operator $\operatorname*{norm}\limits_{i \in I} x_i = \frac{\max\{x_i, 0\}}{\sum\limits_{j \in I} \max\{x_j, 0\}}$ transforms a vector $(x_i)_{i \in I}$ to a discrete distribution, n_{wt} counters represent term frequency of word w in topic t.

Learning of Φ and Θ from (2)–(4) can be done by EM-algorithm, which starts from a random values in Φ and Θ, and iterates E-step (2) and M-steps (3),(4) until convergence. In the sequel we discuss several variations of such EM-algorithm, which are all based on the above formulas but differ in the way how operations are ordered and grouped together.

3 Offline algorithm

The algorithm is given by Fig. 1 (Offline algorithm) and Fig. 2 (ProcessDocument).

Subroutine ProcessDocument(d, Φ) corresponds to equations (2) and (4) from the solution of the ARTM optimization problem (1). ProcessDocument requires a fixed Φ matrix and a vector n_{dw} of term frequencies for a given document $d \in D$, and as a result it returns the topical distribution θ_{td} for the document, and a matrix $(\hat{n}_{wt})$ of size $|d| \times |T|$, where $|d|$ gives the number of distinct terms in the document. The ProcessDocument might be also useful as a separate routine which finds θ_{td} distribution for a new document, but in the Offline algorithm it is rather used as a building block in an iterative EM-algorithm that learns the Φ matrix.

Algorithm 2: ProcessDocument(d, Φ)

Input: document $d \in D$, matrix $\Phi = (\phi_{wt})$;
Output: matrix $(\tilde{n}_{wt})$, vector θ_{td};

1 initialize $\theta_{td} := \frac{1}{|T|}$ for all $t \in T$;
2 **repeat**
3 $\quad p_{tdw} := \underset{t \in T}{\mathrm{norm}}\big(\phi_{wt}\theta_{td}\big)$ for all $w \in d$ and $t \in T$;
4 $\quad \theta_{td} := \underset{t \in T}{\mathrm{norm}}\big(\sum_{w \in d} n_{dw} p_{tdw} + \theta_{td} \frac{\partial R}{\partial \theta_{td}}\big)$ for all $t \in T$;
5 **until** θ_d *converges*;
6 $\tilde{n}_{wt} := n_{dw} p_{tdw}$ for all $w \in d$ and $t \in T$;

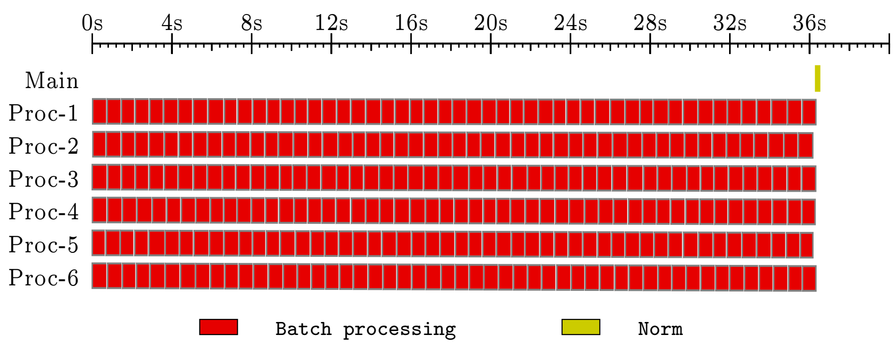

Fig. 1. Gantt chart for execution of the Offline algorithm

The Offline algorithm performs multiple scans over the collection, calls ProcessDocument for each document $d \in D$ from the collection, and then aggregates the resulting $(\hat{n}_{wt})$ matrices into the final (n_{wt}) matrix of size $|W| \times |T|$. After each scan it recalculates Φ matrix according to the equation (3).

At step 2 we split collection D into batches (D_b). This step is not strictly necessary for the Offline algorithm, and it rather reflects an internal implementation detail. For performance reasons the outer loop over batches $b = 1, \ldots, B$ is parallelized across multiple threads, and within each batch the inner loop $d \in D_b$ is executed in a single thread. Each batch is stored in a separate disk file on disk to allow out-of-core streaming of the collection. For typical collections it is reasonable to have around 1000 documents per batch, however for ultimate performance we encourage users to experiment with this parameter. Too small batches can cause disk IO overhead due to lots of small reads, while too large batches will result in bigger tasks that will not be distributed evenly across computation threads.

Note that θ_{td} values appear only within ProcessDocument subroutine. This leads to efficient memory usage because the implementation never stores the entire theta matrix at any given time. Instead, θ_{td} values are recalculated from scratch on every pass through the collection.

Algorithm 3: Online algorithm

Input: collection D, parameters η, τ_0, κ;
Output: matrix $\Phi = (\phi_{wt})$;

create batches $D := D_1 \sqcup D_2 \sqcup \dots \sqcup D_B$;
initialize (ϕ^0_{wt});
for all *update* $i = 1, \dots, \lfloor B/\eta \rfloor$
  $(\hat{n}^i_{wt}) := \mathsf{ProcessBatches}(\{D_{\eta(i-1)+1}, \dots, D_{\eta i}\}, \Phi^{i-1})$;
  $\rho_i := (\tau_0 + i)^{-\kappa}$;
  $(n^i_{wt}) := (1 - \rho_i) \cdot (n^{i-1}_{wt}) + \rho_i \cdot (\hat{n}^i_{wt})$;
  $(\phi^i_{wt}) := \operatorname*{norm}_{w \in W}(n^i_{wt} + \phi^{i-1}_{wt} \frac{\partial R}{\partial \phi_{wt}})$;

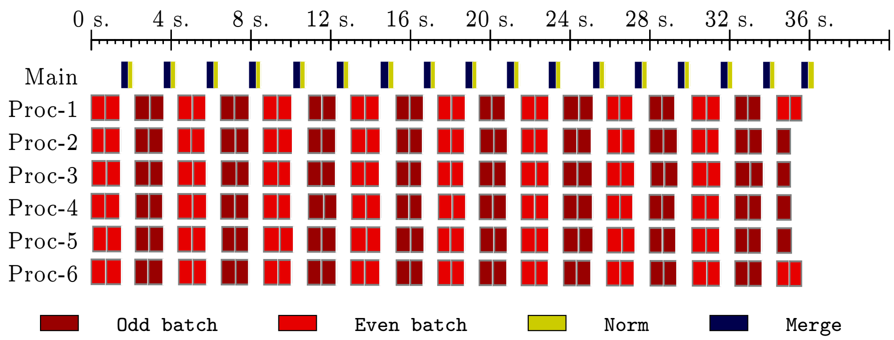

Fig. 2. Gantt chart for new online algorithm

Fig. 1 illustrates a run-time behavior of the Offline algorithm. It shows a Gantt chart, where boxes correspond to the time spent in processing an individual batch. The final box, executed on the main thread, correspond to the time spent in the step 4 to normalize n_{wt} values and produce a new Φ matrix.

4 Online algorithm

The Online algorithm (Fig. 3), originally suggested in [6], improves the convergence rate of the Offline algorithm by re-calculating matrix Φ after every η batches. To simplify the notation we introduce a trivial subroutine

$$\mathsf{ProcessBatches}(\{D_b\}, \Phi) = \sum_{D_b} \sum_{d \in D_b} \mathsf{ProcessDocument}(d, \Phi)$$

that aggregates the output of ProcessDocument across specific set of batches at a constant Φ matrix. The algorithm is then given by Fig. 3. Here the split of the collection $D := D_1 \sqcup D_2 \sqcup \dots \sqcup D_B$ into batches plays far more significant role than in the Offline algorithm, because different splitting algorithmically affects

Algorithm 4: Asynchronous online algorithm

Input: collection D, parameters η, τ_0, κ;
Output: matrix $\Phi = (\phi_{wt})$;

1 create batches $D := D_1 \sqcup D_2 \sqcup \cdots \sqcup D_B$;
2 initialize (ϕ^0_{wt});
3 $F^1 :=$ AsyncProcessBatches$(\{D_1, \ldots, D_\eta\}, \Phi^0)$;
4 **for all** *update* $i = 1, \ldots, \lfloor B/\eta \rfloor$
5   **if** $i \neq \lfloor B/\eta \rfloor$ **then**
6         $F^{i+1} :=$ AsyncProcessBatches$(\{D_{\eta i+1}, \ldots, D_{\eta i+\eta}\}, \Phi^{i-1})$;
7     $\hat{n}^i_{wt} :=$ Await(F^i);
8     $\rho_i := (\tau_0 + i)^{-\kappa}$;
9     $(n^i_{wt}) := (1 - \rho_i) \cdot (n^{i-1}_{wt}) + \rho_i \cdot (\hat{n}^i_{wt})$;
10     $(\phi^i_{wt}) := \operatorname*{norm}_{w \in W}(n^i_{wt} + \phi^{i-1}_{wt} \frac{\partial R}{\partial \phi_{wt}})$;

the result. At step 6 the new n^{i+1}_{wt} values are calculated as a convex combination of the old values n^i_{wt} and the value $\hat{n}^i_{wt}$ produced on the recent batches. Old counters n^i_{wt} are discounted by a factor $(1 - \rho_i)$, which depends on the iteration number. A common strategy is to use $\rho_i = (\tau_0 + i)^{-\kappa}$, where typical values for τ_0 are between 64 and 1024, for κ — between 0.5 and 0.7.

As in the Offline algorithm, the outer loop over batches $D_{\eta(i-1)+1}, \ldots, D_{\eta i}$ is executed concurrently across multiple threads. The problem with this approach is that all threads have no useful work to do during steps 5-7 of the Online algorithm. The threads can not start processing the next batches because a new version of Φ matrix is not ready yet. As a result the CPU utilization stays low, and the run-time Gantt chart of the Online algorithm typically looks like in Fig. 3. The color indicate which version of the p^i_{wt} matrix was used to process each batch (orange for even i, yellow for odd i). Blue box correspond to the time spend in merging n_{wt} with $\hat{n}_{wt}$, Green box is, as before, the time spent to normalize n_{wt} values and produce a new p_{wt} matrix.

In the next section we present an asynchronous non-blocking modification of the online algorithm that results in better CPU utilization.

5 Asynchronous online algorithm

Asynchronous online algorithm is based on two new routines, AsyncProcessBatches and Await. The first one is equivalent to ProcessBatches, except that it just queues the task for an asynchronous execution and returns immediately. Its output is a future object (for example, an std::future from C++11 standard), which can be later passed to Await in order to get the actual result, e.g. in our case the $\hat{n}_{wt}$ values. In between calls to AsyncProcessBatches and Await the algorithm can perform some other useful work, while the background threads are calculating the $\hat{n}_{wt}$ values.

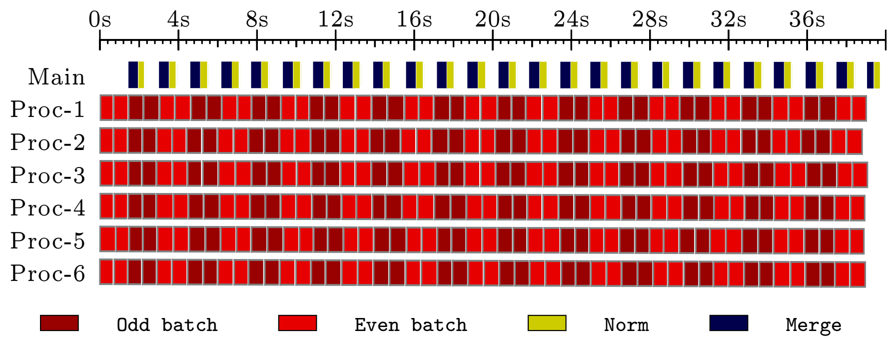

Fig. 3. Gantt chart for async online algorithm

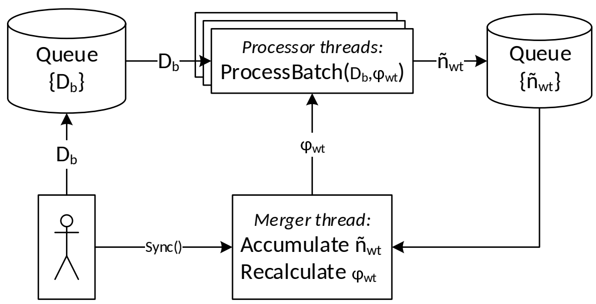

Fig. 4. Diagram of BigARTM components (old architecture)

The resulting algorithm is given by Fig. 4. To calculate $\hat{n}_{wt}^{i+1}$ it uses Φ^{i-1} matrix, which is one generation older than Φ^i matrix used by the conventional Online algorithm 3. This adds an extra "offset" between the moment when Φ matrix is calculated and the moment when it is used, and as a result gives the algorithm additional flexibility to distribute more payload to computation threads. Steps 3 and 5 of the algorithm are just technical tricks to implement the "offset" idea.

Adding an offset should negatively impact the convergence of the asynchronous algorithm 4 comparing to the conventional algorithm **??**. For example, in AsyncProcessBatches the initial matrix Φ^0 is used twice, and the two last matrices $\Phi^{\lfloor B/\eta \rfloor - 1}$ and $\Phi^{\lfloor B/\eta \rfloor}$ will not be used at all. One the other hand the asynchronous algorithm gives better CPU utilization, as clearly shown by the Gantt chart from Fig. 3.

This tradeoff convergence and CPU utilization is evaluated in the experiments from section 7.

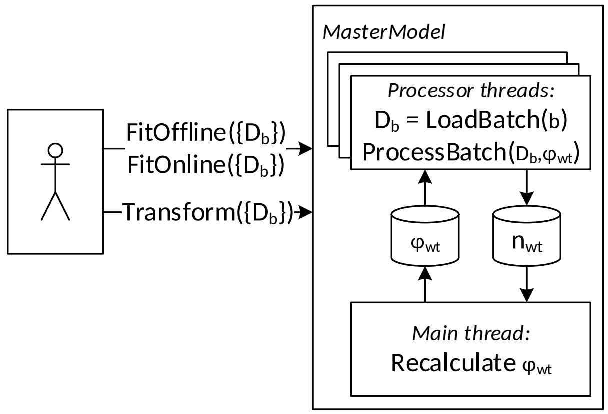

Fig. 5. Diagram of BigARTM components (new architecture)

6 Implementation

The challenging part for the implementation is to aggregate the $\hat{n}_{wt}$ matrices across multiple batches, given that they are processed in different threads. The way BigARTM solves this challenge was changed between versions v0.6 (see Fig. 4) and v0.7 (see Fig. 5). In the old architecture the $\hat{n}_{wt}$ matrices were stored in a queue, and then aggregated by a dedicated *Merger thread*. This often caused performance bottlenecks, particularly because of small batches or due to a small number of iterations in ProcessDocument's inner loop 2-5.

In the new architecture we removed Merger thread, and $\hat{n}_{wt}$ are written directly into the final n_{wt} matrix concurrently from all processor threads. To synchronize the write access we require that no threads simultaneously update the same row in $\hat{n}_{wt}$ matrix, yet the data for distinct words can be written in parallel. This is enforced by spin locks l_w, one per each word in the dictionary W. At the end of ProcessDocument we loop through all $w \in d$, acquire the corresponding lock l_w, append $\hat{n}_{wt}$ to n_{wt} and release the lock. This approach is similar to [13], where the same pattern is used to update a shared stated in a distributed topic modeling architecture.

In our new architecture we also removed *DataLoader* thread, which previously was loading batches from disk. Now this happens directly from processor thread, which simplified the architecture without sacrificing performance.

In addition, we provided a cleaner API so now the users may use simple FitOffline, FitOnline methods to learn the model, and Transform to apply the model to the data. Previously the users had to interact with low-level building blocks, such as ProcessBatches routine.

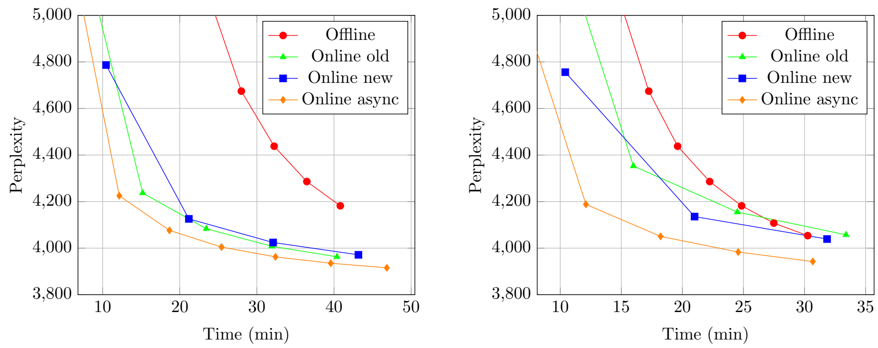

Fig. 6. Perplexity versus time for Wikipedia (left) and Pubmed (right), $|T| = 100$ topics

7 Experiments

In this section we compare the effectiveness of the Offline algorithm (Fig. 1), Online algorithm (Fig. 3), asynchronous online algorithm (Fig. 4), and the old non-deterministic online algorithm from BigARTM v0.6, described in [17]. According to [17], the later algorithm runs ca. 10 times comparing to Gensim [10], and twice faster comparing to Vowpal Wabbit (VW) [11] in a single thread; an with multiple threads BigARTM wins even more.

In the experiments we use *Wikipedia* dataset ($|D| = 3.7m$ articles, $|W| = 100k$ words) and *Pubmed* dataset ($|D| = 8.2m$ abstracts, $|W| = 141k$ words). The experiments ran on Intel Xeon CPU E5-2650 v2 system with 2 processors, 16 physical cores in total (32 with hyper-threading).

Fig. 6 show the *perplexity* as a function of the time spend by the four algorithms listed above. The perplexity measure is defined as

$$\mathscr{P}(D, p) = \exp\left(-\frac{1}{n} \sum_{d \in D} \sum_{w \in d} n_{dw} \ln \sum_{t \in T} \phi_{wt} \theta_{td}\right), \tag{5}$$

where $n = \sum_d n_d$. Lower perplexity means better result. Each point on the figures corresponds to a moment when the algorithm finishes a complete scan of the collection. Each algorithm was time-boxed to run for a 40 minutes for Wikipedia and 30 minutes for Pubmed.

Table 1 gives peak memory usage for $|T| = 1000$ topics model on Wikipedia and Pubmed datasets. TDB: explain memory usage (how many matrices, what is the size of each matrix, how big batches affect memory usage, why old algorithm is so inefficient)

8 Conclusions

TBD

Table 1. BigARTM peak memory usage in GB, $|T| = 1000$ topics

	Offline	Online	Async	Old online
Pubmed	5.17	4.68	8.18	13.4
Wiki	1.74	2.44	3.93	7.9

Acknowledgements. The work was supported by Russian Science Foundation (grant 15-18-00091).

References

1. D. M. Blei. Probabilistic topic models. *Communications of the ACM*, 55(4):77–84, 2012.
2. Daud, A., Li, J., Zhou, L., Muhammad, F.: Knowledge discovery through directed probabilistic topic models: a survey. Frontiers of Computer Science in China 4(2), 280–301 (2010)
3. J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.Y. Liu, and W. Y. Ma. LightLDA: Big Topic Models on Modest Computer Clusters. In Proceedings of the 24th International Conference on World Wide Web, pp. 1351-1361, 2015.
4. D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
5. T. Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 50–57, 1999.
6. M. D. Hoffman, D. M. Blei, and F. R. Bach. Online learning for latent dirichlet allocation. In *NIPS*, pages 856–864. Curran Associates, Inc., 2010.
7. D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed algorithms for topic models. *J. Mach. Learn. Res.*, 10:1801–1828, Dec. 2009.
8. Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang. PLDA: Parallel latent Dirichlet allocation for large-scale applications. In *Proceedings of the 5th International Conference on Algorithmic Aspects in Information and Management*, pages 301–314, 2009.
9. Z. Liu, Y. Zhang, E. Y. Chang, and M. Sun. PLDA+: parallel latent Dirichlet allocation with data placement and pipeline processing. *ACM Trans. Intell. Syst. Technol.*, 2(3):26:1–26:18, May 2011.
10. R. Řehůřek and P. Sojka. Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010.
11. J. Langford, L. Li, and A. Strehl. Vowpal wabbit open source project. *Technical report*, Yahoo!, 2007.
12. A. K. McCallum, A Machine Learning for Language Toolkit. *http://mallet.cs.umass.edu*, 2002.
13. A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1-2):703–710, Sept. 2010.
14. K. V. Vorontsov. Additive regularization for topic models of text collections. *Doklady Mathematics*, 89(3):301–304, 2014.

15. K. V. Vorontsov and A. A. Potapenko. Additive regularization of topic models. *Machine Learning, Special Issue on Data Analysis and Intelligent Optimization*, 2014.
16. K. V. Vorontsov and A. A. Potapenko. Tutorial on probabilistic topic modeling: Additive regularization for stochastic matrix factorization. In *AIST'2014, Analysis of Images, Social networks and Texts*, volume 436, pages 29–46. Springer International Publishing Switzerland, Communications in Computer and Information Science (CCIS), 2014.
17. K. Vorontsov, O. Frei, M. Apishev, P. Romov and M. Dudarenko. BigARTM: Open Source Library for Regularized Multimodal Topic Modeling of Large Collections. In *AIST'2015, Analysis of Images, Social networks and Texts*, volume 542, pages 370–381. Springer International Publishing Switzerland, Communications in Computer and Information Science (CCIS), 2015.

Appendix

Fig. 7 illustrates run-time behavior of the old online algorithm, implemented in previous BigARTM version (v0.6).

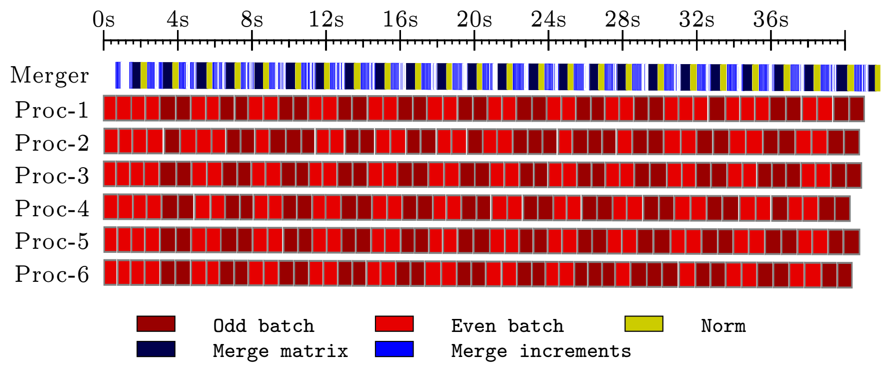

Fig. 7. Gantt chart for execution of the old online algorithm from BigARTM v0.6

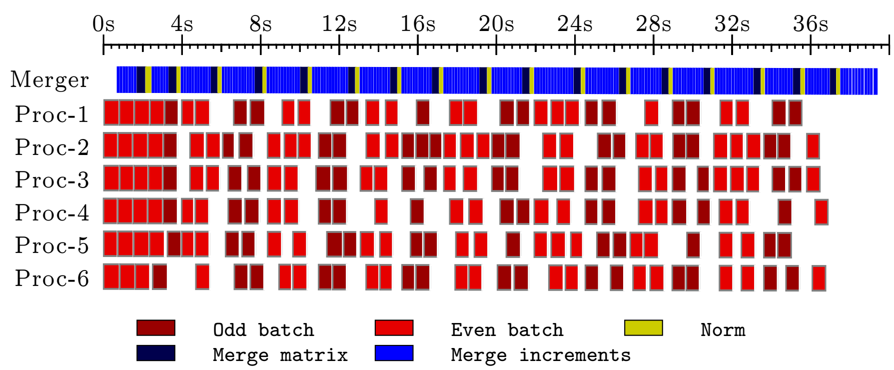

Fig. 8. Gantt chart for execution of the old online algorithm from BigARTM v0.6